Introduction

Python is a general-purpose programming language on its own, but with the help of a few popular libraries (numpy, scipy, matplotlib) it becomes a powerful environment for scientific computing.

Before going through this tutorial, we expect you have some experience with Python and numpy.For whom has no knowledge or experience, you can go through this video: Numpy Tutorial

In this tutorial, we will cover:

- Basic Python: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Functions, Classes
- Numpy: Arrays, Array indexing, Datatypes, Array math, Broadcasting
- Matplotlib: Plotting, Subplots, Images
- IPython: Creating notebooks, Typical workflows

```
# check python version
# for those who run python with their local machine, I recommend to
use
# 1 python 3.7 or python 3.7.12 or 3.8.12
# (https://www.python.org/downloads/windows/)
# (https://www.python.org/downloads/macos/)
# use "CTRL+ / " to comment / uncomment an line in Colab

!python --version
```

```
Python 3.7.12
```

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable.

An implementation of the classic quicksort algorithm in Python:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

array_to_sort = [3,6,8,10,1,2,1]
print(quicksort(array_to_sort))
```

```
[1, 1, 2, 3, 6, 8, 10]
```

Basic data types

```
#numeric

x = 3
print(f'data type {x}, {type(x)}')

# numeric operations
print(x + 1)   # Addition
print(x - 1)   # Subtraction
print(x * 2)   # Multiplication
print(x ** 2)  # Exponentiation

x += 1
print(x)
x *= 2
print(x)

y = 2.5
print(f'type cast by value: {type(y)}')
print(y, y + 1, y * 2, y ** 2)
```

```
data type 3, <class 'int'>
4
2
6
9
4
8
type cast by value: <class 'float'>
2.5 3.5 5.0 6.25
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x--) operators.

Python also has built-in types for long integers and complex numbers; you can find all of the details in the documentation.

```
# Boolean

t, f = True, False
print(type(t))

print(t and f) # Logical AND;
print(t or f)  # Logical OR;
print(not t)   # Logical NOT;
print(t != f)  # Logical XOR;
```

```
<class 'bool'>
False
True
```

```
False
True
```

```
# Strings

hello = 'hello'   # String literals can use single quotes
world = "world"   # or double quotes; it does not matter
print(hello, len(hello))

hw = hello + ' ' + world  # String concatenation
print(hw)

hw12 = '{} {} {}'.format(hello, world, 12)  # string formatting
print(hw12)

# f string
print(f'print f string: {hello} and {world}')
```

```
hello 5
hello world
hello world 12
print f string: hello and world
```

```
s = "hello"
print(s.capitalize())  # Capitalize a string
print(s.upper())       # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces
print(s.center(7))     # Center a string, padding with spaces
print(s.replace('l', '(***)'))  # Replace all instances of one 
substring with another

s2 = "   hello, world"
print(s2)
print("after calling strip:", s2.strip())  # Strip leading and 
trailing whitespace
```

```
Hello
HELLO
  hello
 hello
he(***)(***)o
   hello, world
after calling strip: hello, world
```

Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples.

```
# List
```

```
xs = [3, 1, 2]   # Create a list
print(xs, xs[2])
print(xs[-1])     # Negative indices count from the end of the list as
-1; prints "2"
```

```
[3, 1, 2] 2
2
```

```
xs = [3, 1, 2]   # reset the list

xs[2] = 'a string'    # Lists can contain elements of different types
print(xs)

xs.append('new item') # Add a new element to the end of the list
print(xs)

x = xs.pop()     # Remove and return the last element of the list
print(x)
print(f'after pop: {xs}')
```

```
[3, 1, 'a string']
[3, 1, 'a string', 'new item']
new item
after pop: [3, 1, 'a string']
```

```
# Slicing

nums = list(range(5))    # range is a built-in function that creates a
list of integers
print(nums)         # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])    # Get a slice from index 2 to 4 (exclusive);
prints "[2, 3]" - index includes the lower, excludes the upper
print(nums[2:])     # Get a slice from index 2 to the end; prints "[2,
3, 4]"
print(nums[:2])     # Get a slice from the start to index 2
(exclusive); prints "[0, 1]"
print(nums[:])      # Get a slice of the whole list; prints ["0, 1, 2,
3, 4]"
print(nums[:-1])    # Slice indices can be negative; prints ["0, 1, 2,
3]"
nums[2:4] = [8, 9] # Assign a new sublist to a slice
print(nums)         # Prints "[0, 1, 8, 9, 4]"
```

```
[0, 1, 2, 3, 4]
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

Iterations

You can loop over the elements of a list like this

```
fruits = ['apple', 'banana', 'grapefruit', 'orange']
for fruit in fruits:
    print(fruit)
```

```
apple
banana
grapefruit
orange
```

If you want access to the index of each element within the body of a loop, use the built-in enumerate function:

```
fruits = ['apple', 'banana', 'grapefruit', 'orange']
for idx, fruit in enumerate(fruits):
    print('#{}: {}'.format(idx + 1, fruit))
```

```
#1: apple
#2: banana
#3: grapefruit
#4: orange
```

List comprehensions:

When programming, frequently we want to transform one type of data into another. For example, the primary code to compute square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)
```

```
[0, 1, 4, 9, 16]
```

The advanced way for using list comprehension is:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)
```

```
[0, 1, 4, 9, 16]
```

List comprehensions with conditions:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]  # 3 is skipped
because 3 % 2 == 1
print(even_squares)
```

```
[0, 4, 16]
```

Dictionaries

A dictionary stores (key, value) pairs, similar to a Map in Java or an object in Javascript.

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with
some data
print(d['cat'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints
"True"
```

```
cute
True
```

```
d['fish'] = 'wet'    # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
```

```
wet
```

```
# KeyError: 'monkey' not a key of d
# print(d['monkey'])
```

```
---------------------------------------------------------------------
-----
KeyError                                  Traceback (most recent call
last)
<ipython-input-20-78fc9745d9cf> in <module>()
----> 1 print(d['monkey'])  # KeyError: 'monkey' not a key of d

KeyError: 'monkey'
```

```
print(d.get('monkey', 'N/A'))  # Get an element with a default; prints
"N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default; prints
"wet"
```

```
N/A
wet
```

```
del d['fish']        # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

```
N/A
```

It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for key, values in d.items():
    print(f'{key} has {values} legs')
```

```
person has 2 legs
cat has 4 legs
spider has 8 legs
```

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries.

```
nums = [0, 1, 2, 3, 4, 5, 6]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}  # 3 and
5 will be skipped
print(even_num_to_square)
```

```
{0: 0, 2: 4, 4: 16, 6: 36}
```

Sets

A set is an unordered collection of distinct elements. When we convert a list to a set, we can remove all duplicate elements in the list

```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints
"True"
print('fish' in animals)  # prints "False"
```

```
True
False
```

```
animals.add('fish')      # Add an element to a set
print('fish' in animals)
print(len(animals))       # Number of elements in a set;
```

```
True
3
```

```
animals.add('cat')       # Adding an element that is already in the
set does nothing
print(len(animals))
animals.remove('cat')    # Remove an element from a set
print(len(animals))
```

```
3
2
```

```
fruits = ['apple', 'banana', 'orange', 'apple', 'orange']
print(f'the list has {fruits}')
fruits = set(fruits)
print(f'the set has {fruits}')
```

```
the list has ['apple', 'banana', 'orange', 'apple', 'orange']
the set has {'apple', 'orange', 'banana'}
```

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#{}: {}'.format(idx + 1, animal))
```

```
#1: cat
#2: fish
#3: dog
```

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

```
from math import sqrt
print('List: ', [int(sqrt(x)) for x in range(30)])
print('Set: ', {int(sqrt(x)) for x in range(30)})
```

```
List:  [0, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 
4, 4, 4, 4, 5, 5, 5, 5, 5]
Set:  {0, 1, 2, 3, 4, 5}
```

Tuples

A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot.

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with 
tuple keys
print(d)
```

```
{(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4, (5, 6): 5, (6, 
7): 6, (7, 8): 7, (8, 9): 8, (9, 10): 9}
```

```
t = (5, 6)       # Create a tuple
print(type(t))
print(d[t])
print(d[(1, 2)])
# tuple is immutable, the items inside cannot be re-assigned
# this line will return an error
# t[0] = 1
```

```
<class 'tuple'>
5
1
```

```
# tuple is commonly used to return more than one values from a 
function
```

```
def square_sqrt(val):
  if val > 0:
    return val**2, sqrt(val)
  else:
    return 0

vals = square_sqrt(25)
print(f'data type: {type(vals)}, value: {vals}')
squares, square_root = square_sqrt(36)  # unpack the tuple by
assignment
print('square = {}, square root = {}'.format(squares, square_root))
```

```
data type: <class 'tuple'>, value: (625, 5.0)
square = 1296, square root = 6.0
```

Functions

Python functions are defined using the def keyword.

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
```

```
negative
zero
positive
```

We will often define functions to take optional keyword arguments.

```
def hello(name, loud=False):
    if loud:
        print('HELLO, {}'.format(name.upper()))
    else:
        print('Hello, {}!'.format(name))

hello('Bob')
hello('Fred', loud=True)  #
hello('Stanley', 1)  # implicit pass
```

```
Hello, Bob!
HELLO, FRED
HELLO, STANLEY
```

Classes

Define a python class

```
class Item:
    # class level static attribute
    pay_rate = 0.8 # The pay rate after 20% off

    def __init__(self, name: str, price: float, quantity=0):
        # Run validations to the received arguments
        assert price >= 0, f"Price {price} is not greater than or 
equal to zero!"
        assert quantity >= 0, f"Quantity {quantity} is not greater or 
equal to zero!"

        # Assign to self object
        self.__name = name
        self.price = price
        self.quantity = quantity

    @property
    # property decorator - Read-Only Attribute
    def name(self):
      return self.__name

    @name.setter
    def name(self, value):
      if len(value) > 15:
        raise Exception("The name cannot exceed 15 characters.")
      else:
        self.__name =value


    def calculate_total_price(self):
        return self.price * self.quantity

    def apply_discount(self):
        self.price = self.price * self.pay_rate

    # define the default method for the instance representation string
    def __repr__(self):
        return f"Item('{self.name}', {self.price}, {self.quantity})"

item1 = Item("Phone", 100, 1)
item2 = Item("Laptop", 1000, 3)
item3 = Item("Cable", 10, 5)
item4 = Item("Mouse", 50, 5)
item5 = Item("Keyboard", 75, 5)
items = [item1, item2, item3, item4, item5]
```

```
for i in items:
  print(i)  # print the return string defined by  __repr__
```

```
Item('Phone', 100, 1)
Item('Laptop', 1000, 3)
Item('Cable', 10, 5)
Item('Mouse', 50, 5)
Item('Keyboard', 75, 5)
```

```
# call the instance method
for i in items:
  print(f'original total: {i.calculate_total_price()}')
  i.apply_discount()
print('---after applied discount----')
for i in items:
  print(f'discount total: {i.calculate_total_price()}')
```

```
original total: 100
original total: 3000
original total: 50
original total: 250
original total: 375
---after applied discount----
discount total: 80.0
discount total: 2400.0
discount total: 40.0
discount total: 200.0
discount total: 300.0
```

```
# show the properties of the instance
vars(item1)
```

```
{'_Item__name': 'Phone', 'price': 80.0, 'quantity': 1}
```

```
# use the getter to access __name
for i in items:
  print(i.name)
  # print(i.__name) # cannot access directly

# use the setter to change __name
item1.name = 'iPhone 13'
item2.name = 'Mac Air'
print('----after name changes----')
for i in items:
  print(i.name)
```

```
Phone
Laptop
Cable
Mouse
```

```
Keyboard
----after name changes----
iPhone 13
Mac Air
Cable
Mouse
Keyboard
```

Class Inheritance

```
# sub-class -- a Phone class inherited from Item class, add a
broken_phone property to indicate the # of broken phones
class Phone(Item):
  all = []
  def __init__(self, name: str, price: float, quantity=0,
broken_phones=0):
    super(Phone, self).__init__(name=name, price=price,
quantity=quantity) # instantiate the super class object
    # Run validations to the received arguments
    assert broken_phones >= 0, f"Broken Phones {broken_phones} is not
greater or equal to zero!"
    self.broken_phones = broken_phones
    Phone.all.append(self)

  def total_price_exclude_broken(self):
     return self.price * (self.quantity - self.broken_phones)

  def __repr__(self):
     return f"{self.__class__.__name__} -- ('{self.name}',
{self.price}, {self.quantity}, {self.broken_phones})"

phone1 = Phone("iPhone 12", 899, 2, 1)
phone2 = Phone("iPhone 11", 699, 10, 3)
phone3 = Phone("iPhone 13 mini", 999, 15)

phones = [phone1, phone2, phone3]

for p in phones:
  print(p)
  print(f'with broken phones: {p.calculate_total_price()}')

print()
print('----Value excluding broken phones----', '\n')

for p in phones:
  print(f'Exclude broken phones: {p.total_price_exclude_broken()}')
```

```
Phone -- ('iPhone 12', 899, 2, 1)
with broken phones: 1798
Phone -- ('iPhone 11', 699, 10, 3)
```

```
with broken phones: 6990
Phone -- ('iPhone 13 mini', 999, 15, 0)
with broken phones: 14985

----Value excluding broken phones----

Exclude broken phones: 899
Exclude broken phones: 4893
Exclude broken phones: 14985
```

Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

```
# import the numpy package

import numpy as np
```

Numpy Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
# Create a rank 1 array
a = np.array([1, 2, 3])
print(type(a), a.shape, a[0], a[1], a[2])

# Change an element of the array
a[0] = 5
print(a)

# Create a rank 2 array
b = np.array([[1,2,3],[4,5,6]])
print(b)

print(b.shape)
# numpy array slicing
print(b[0, 0], b[0, 1], b[1, 0])
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
[[1 2 3]
 [4 5 6]]
```

```
(2, 3)
1 2 4
```

Numpy also provides many functions to create arrays

Good for tensor operations

- first - set up the tensor shape
- second - fill / compute the values

```
# Create an array of all zeros
a = np.zeros((2,2))  # Create an array of all zeros
print(a)

# Create an array of all ones
b = np.ones((1,2))

# Create a constant array
c = np.full((2,2), 7)
print(c)

print()
# Create a 2x2 identity matrix
d = np.eye(2)
print(d)

print()
# Create a 3x3 identity matrix
e = np.eye(3)
print(e)

print()
# Create an array filled with random values
e = np.random.random((3,3)) # with mean=0
print(e)
```

```
[[0. 0.]
 [0. 0.]]
[[7 7]
 [7 7]]

[[1. 0.]
 [0. 1.]]

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

[[0.31914832 0.18614806 0.19662313]
 [0.01001235 0.34407035 0.80764068]
 [0.63783024 0.42018236 0.01456189]]
```

Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print('The whole matrix')
print(a)
print()
# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
# slice row 0-1 (exclude row 2), column 1-2 (exclude column 0 and 3)
b = a[:2, 1:3]
print('The sliced part:')
print(b)
```

```
The whole matrix
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

The sliced part:
[[2 3]
 [6 7]]
```

```
print(a[0, 1])
print(a)
b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])
print(a)
```

```
2
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
77
[[ 1 77  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array.

```
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
print()

row_r1 = a[1, :]    # Rank 1 view of the second row of a: rank 2 -->
rank 1
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a: keep as rank
2
row_r3 = a[[1], :]  # Rank 2 view of the second row of a: keep as rank
2
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
print(row_r3, row_r3.shape)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[[5 6 7 8]] (1, 4)
```

```
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]  # rank 1
col_r2 = a[:, 1:2] # rank 2
print(col_r1, col_r1.shape)
print()
print(col_r2, col_r2.shape)
```

```
[ 2  6 10] (3,)

[[ 2]
 [ 6]
 [10]] (3, 1)
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array.

```
a = np.array([[1,2], [3, 4], [5, 6]])
print(a)
print()
# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]]) # slice by row-col indexing

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

```
[[1 2]
 [3 4]
 [5 6]]

[1 4 5]
[1 4 5]
```

```python
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))
```

```
[2 2]
[2 2]
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

```python
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
print()

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

[ 1  6  7 11]
```

```python
# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10
print(a)
```

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])
print(a)
print()

bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                    # this returns a numpy array of Booleans of the
same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print(bool_idx)
```

```
[[1 2]
 [3 4]
 [5 6]]

[[False False]
 [ True  True]
 [ True  True]]
```

```
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a)
print(bool_idx)
print(a[bool_idx]) # pick the elements from array a with
boolean==True, rank 2 --> rank 1

# We can do all of the above in a single concise statement:
print(a[a > 2])
```

```
[[1 2]
 [3 4]
 [5 6]]
[[False False]
 [ True  True]
 [ True  True]]
[3 4 5 6]
[3 4 5 6]
```

Data Type

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.

```
x = np.array([1, 2])  # Let numpy choose the datatype
y = np.array([1.0, 2.0])  # Let numpy choose the datatype
```

```
z = np.array([1.0, 2.0], dtype=np.int64)  # Force a particular
datatype

print(x.dtype, y.dtype, z.dtype)
```

```
int64 float64 int64
```

Numpy Array Math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

```
# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

```
# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Note that the "*" operator is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

Use the "@" operator in numpy as dot product operation

```
print(v @ w)
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

```
219
[29 67]
[29 67]
[29 67]
```

Numpy provides many useful functions for performing computations on arrays.

```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
print()

# transpose
print(x)
print("transpose\n", x.T)
```

```
10
[4 6]
[3 7]

[[1 2]
 [3 4]]
transpose
 [[1 3]
 [2 4]]
```

Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

```
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as
x
print('---the original matrix ----')
print(x)
print('----rank 1 vector to be added----')
print(v)
# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print('----new matrix----')
print(y)
```

```
---the original matrix ----
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
----rank 1 vector to be added----
[1 0 1]
----new matrix----
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

However when the matrix x is very large, computing an explicit loop in Python could be slow.

To reduce the runtime, we consider adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing elementwise summation of x and vv. We could implement this approach like this:

```
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
print(vv)                # Prints "[[1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

```
y = x + vv  # Add x and vv elementwise
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:

```
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if v actually had shape `(4, 3)`, where each row was a copy of v, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.

4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation from the documentation or this explanation.

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

We provide some examples of broadcasting below:

```
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

```
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:

print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

```
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:

print((x.T + w).T)
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

```
# Another solution is to reshape w to be a row vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

```
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the matplotlib.pyplot module.

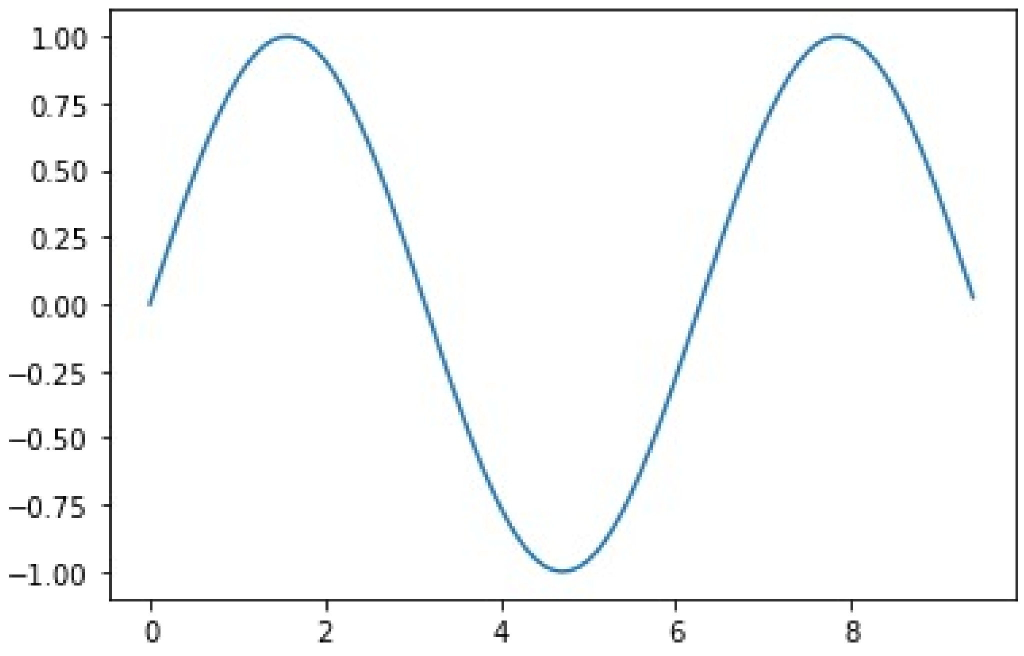
```
import matplotlib.pyplot as plt
# to run this special iPython command, we will be displaying plots
inline:
%matplotlib inline
```

Plotting

```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
```

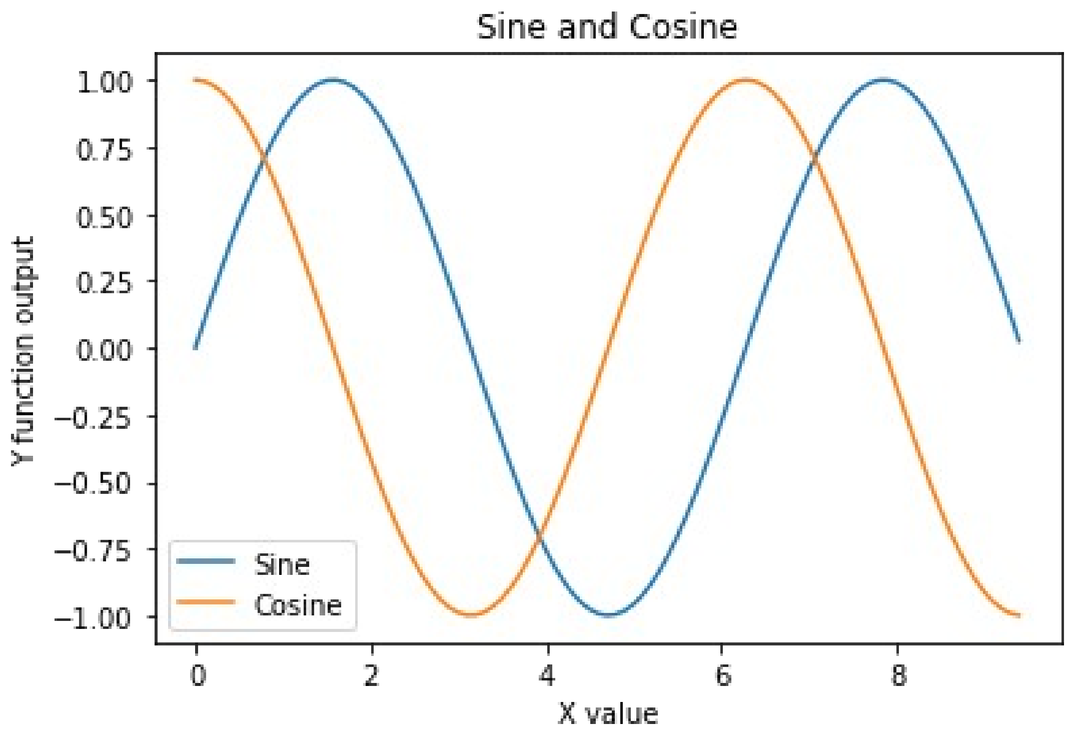
```
[<matplotlib.lines.Line2D at 0x7f28fde3bb90>]
```



```
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('X value')
plt.ylabel('Y function output')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
```

```
<matplotlib.legend.Legend at 0x7f28fd92fe90>
```



Subplots

You can plot different things in the same figure using the subplot function.

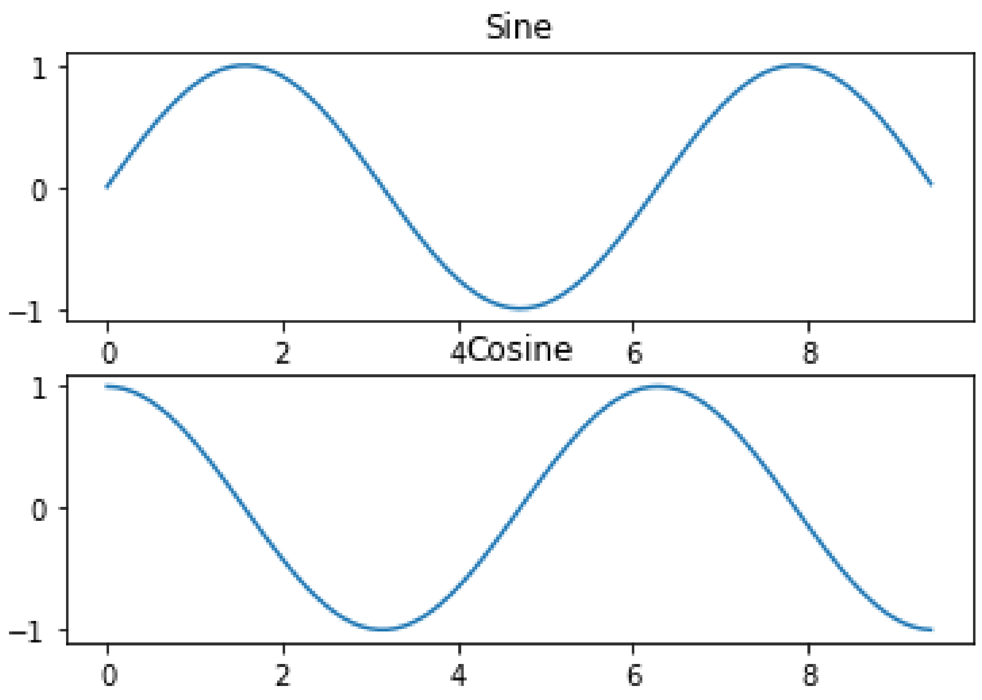
```
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```


Sine
1
0
−1
0
2
4
6
8
Cosine
1
0
−1
0
2
4
6
8